

PROGRAMMING EXPERT

FAQ

Q How do I go about debugging a program?

A Bugs happen when your program does something you didn't expect. Often the difference between your plan and reality boils down to the contents of a variable. If a variable contains the wrong value – or no value at all – you can expect the unexpected.

The first step to effective debugging is to understand your program well enough to know what should be in each variable at each moment. Next, you must find out what the variables really contain. To do this you use a debugger, which monitors your program closely as it is run.

Debuggers also let you execute a program line by line, allowing you to check that it executes the instructions in the order you expect. When you find a discrepancy, you've located the bug.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk



Sniffing network traffic

To identify what's being sent over your network connection, you need a packet sniffer. Mike James shows you how to build one and get to the bottom of those mystery data packets



CODE FOR THE FINISHED PRODUCT



Do you know what data is being sent over your network connection? The chances are you don't, and it's quite difficult to find out.

This month's project is all about reading and analysing the data packets that pass your network adaptor. The program we'll build is called a packet sniffer. This project takes us into some interesting areas of .NET. We'll work with low-level network sockets, decode raw IP packets, use generic classes and create an advanced multithreaded program without really trying.

To begin, get hold of a copy of Visual C# Express Edition, which is free from the Microsoft website, and start a new Windows application project called Sniffy.

RAW SOCKETS

The .NET framework provides lots of standard classes that make using network facilities very easy. For instance, if you want to make a TCP connection to download a webpage, there are custom classes that do this very job. Creating a packet sniffer is a little more low level but it's still just a matter of using a suitable .NET class – in our case, the Socket class.

The Socket class is an object-oriented wrapper around Windows's Socket API. Sockets were invented by researchers at UC Berkeley in 1989 as a way of simplifying communications. The Windows implementation is called WinSock, but the .NET framework spares us from having to get involved in calling its functions directly.

Place a button on the form of the new Windows project. This will be used to start sampling the IP traffic on a particular network adaptor. To make the user interface cleaner, we are going to implement a toggle button; its caption begins as Start, changes to Cancel as soon as it's clicked, goes back to Start when it's clicked again, and so on. Set the button's text property to Start and the button's click handler to:

```
private void button1_Click(
    object sender, EventArgs e)
{
    if (button1.Text == "Start")
    {
        button1.Text = "Cancel";
    }
}
```

We'll deal with the Else part of the If statement a little later. Next we need to create a socket and other classes relating to networking. For easy access to the relevant objects, add the following to the start of the code:

```
using System.Net;
using System.Net.Sockets;
```

The socket needs to be a global object, so you must also add the following to the start of the code:

```
private Socket sock;
```

Then add the following to the button's click event handler to create the socket:

```
sock = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Raw,
    ProtocolType.IP);
```

A socket can work with almost any type of network. InterNetwork specifies that Internet Protocol version 4 (IPv4) addresses be used. Raw indicates that we want to work with all packets, even admin and control packets that are normally seen only by the operating system. Finally, IP specifies that the IPv4 protocol is in use.

A socket has to be 'bound' to a particular IP address – this is the address it represents on the network. The socket sends data from this address and receives all

traffic to this address. In this case, we need the IP address of the main network adaptor in the computer. We'll put a text box on the form to allow the user to enter this, and use its contents to build first an IPAddress object and then an IPEndPoint object:

```
sock.Bind(new IPEndPoint(
    IPAddress.Parse(textBox1.Text),
    0));
```

The final parameter specifies the port that the socket is using. 0 means 'work with all ports'. We now have a socket ready to do some work, but because we want to implement a raw socket there are some final parameters to set:

```
sock.SetSocketOption(
    SocketOptionLevel.IP,
    SocketOptionName.HeaderIncluded,
    true);
```

This ensures that we get the IP header complete with the From and To addresses, as well as the data part of the packet. To tell the socket and the network adaptor we want to receive all packets, no matter what type, we have to use a method that links to a cryptic Sockets API called IOControl, or IOCTL for short. The Socket class hasn't made this API any easier to use, apart from spelling out its name in full:

```
byte[] bIn = { 1, 0, 0, 0 };
byte[] bOut = new byte[4];
sock.IOControl(
    IOControlCode.ReceiveAll,
    bIn, bOut);
```

There is no way to make this look any more understandable; you just have to look up which control codes are needed in the two parameter arrays and use them. Now the raw socket is fully prepared and ready for action.

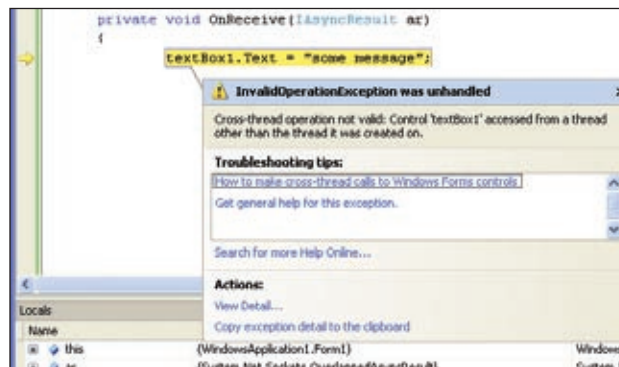
TIME AND THREADS

The next design problem is how to control the duration of the packet sniffing. Taking a timed sample of packets seems like a good approach. The user can still terminate the sample before the time is up by clicking the button. Add a timer, a text box for the sample time and some suitable labels to the form. We then simply set the timer going, using the contents of the text box to set the timer interval:

```
timer1.Interval = Int32.Parse(
    textBox2.Text) * 1000;
timer1.Start();
```

When the timer interval is up, the data gathered by the socket will be processed, but first we have to start the data gathering. We could use the Receive method to get the data but this is a blocking method, which means that when you call it, it doesn't return any data until a packet is received. So it blocks the thread of execution used to call it from doing anything else. Consequently, doing this in the Start/Cancel button's click event handler would block the application's main thread. Until unblocked, the interface would ignore the user, no matter how hard they hit the Cancel button.

This clearly isn't a good idea, but the alternative – to explicitly use multiple threads – is complicated. A simpler way is to use the



◀ Oops: a cross-threaded application

Socket's BeginReceive and EndReceive methods, which implement an asynchronous data read that doesn't block the calling thread. BeginReceive starts the receive operation using a new thread, and when data is ready it calls a delegate – a function you designate – but again using the new thread. This is an easy way to create a multithreaded application; however, it is so easy that you won't realise what you've done and you may introduce subtle errors. It's worth spending some time understanding how the asynchronous call using BeginReceive works; it's an example of a more general asynchronous call facility in .NET.

To get the asynchronous socket read started, we just have to call BeginReceive:

```
sock.BeginReceive(buf, 0,
    buf.Length,
    SocketFlags.None,
    new AsyncCallback(OnReceive),
    null);
```

We still have to supply a byte array buffer, which must be a global variable so other parts of the program can access it:

```
private byte[] buf = new byte[4096];
```

We also have to supply the OnReceive function, which is the delegate called when the read is complete and the buffer has the packet data.

The call to BeginReceive starts the reception of the first packet; reception of subsequent packets is initiated in the OnReceive function. This continues until the user clicks Cancel or the timer expires, triggering the Tick event handler. The 'else' clause of our original If statement simply has to stop the timer and perform a manual call of the event handler:

```
else
{
    timer1.Stop();
    timer1_Tick(new object(),
        new EventArgs());
}
```

This completes the button's click event handler. Next we have to develop the OnReceive function and the Timer's Tick event handler.

DICTIONARIES AND ONRECEIVE

The OnReceive function will be called by the socket using a different thread from the one that runs the rest of the program and we must be careful not to create bugs due to two threads trying to access the same resource at the same

time. The .NET Framework does its best to keep us out of trouble by detecting and generating an error when a thread that didn't create a control tries to access the control.

Say you try a command such as:

```
textBox1.Text = "some message";
```

In OnReceive you will get an error message telling you that you have attempted a cross-threaded operation. You can perform cross-thread access to controls without generating an error, but it's a lot of trouble and best avoided unless absolutely necessary.

You can access data structures and objects from another thread, but be careful. Within the OnReceive function, we need to store the information of interest from the packet as quickly as possible so we can start to receive another packet before it is missed. There are several ways of doing this (an array, for example) but there are many advantages to using a Dictionary object. This stores and retrieves data using a key – in this case, the From address. A Dictionary object can store and retrieve values based on a key much faster than trying to find something in an array. It is also available in .NET in a new and easier form: a generic Dictionary object that can work with almost any data type.

A generic class or generic method is one that has additional parameters to specify the data type it will deal with. For example, to declare a Dictionary object that uses an Int32 as the key and stores an Int32 array, you would write:

```
private Dictionary<UInt32, UInt32[]>
    RawPackets = new Dictionary<UInt32,
        UInt32[]>(256);
```

Notice the use of pointed brackets to enclose the type parameters. The first parameter specifies the key type and the second the value type. This is the Dictionary object we need to store the IP addresses, so RawPackets should be added to the global definitions.

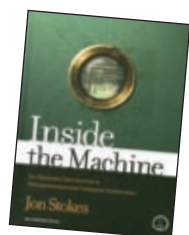
The job of the OnReceive function is to transfer the data we want from the packet to the RawPackets Dictionary object. There is a lot of information in the packet header so to keep things simple, we'll extract just a few items. If you want to see a full definition, look it up using Wikipedia (<http://en.wikipedia.org/wiki/Packet>). The OnReceive function starts with the EndReceive method with which every BeginReceive has to be paired:

```
private void OnReceive(IAsyncResult
    ar)
{
    try
```

NO STARCH PRESS Inside the Machine

★★★★★

£22.50



Books on what makes PCs tick are becoming increasingly rare. You can live in ignorance of how they work if you just learn what to do. But programmers and even advanced computer users will benefit from learning how everything works, and Jon Stokes' enthusiastic account is a good read.

This is a colourful discussion of the general principles behind processors with lots of specifics about the Pentium Pro, the Pentium 4 and PowerPC, plus minor details on many others. It covers pipelines, caches, dual core and so on, but it doesn't cover the wider system architecture so there is no Northbridge and Southbridge coverage. This isn't too unreasonable, as it's just a short book. Lots of full-colour diagrams help make processor operation seem simple. Less effective are the explanations in terms of how a car production line works; explanations in terms of how a processor works would be better.

If you want to know how real processors make your applications come to life, this is one of a handful of decent books. It's fun, if not a practical necessity.

SUMMARY

- ☒ Enthusiastic approach
- ☒ Poor metaphors don't help

SUPPLIER www.amazon.co.uk
ISBN 1593271042
PAGES 432

ADDISON-WESLEY Start-to-Finish Visual Basic 2005

★★

£24



This is a book written by an expert who has no idea how to communicate the ideas that he lives with every day. It's supposed to be for the beginner and expert alike, but the start of the book crams in so much irrelevant detail that beginners aren't going to get very much from it.

Tim Patrick clearly knows his stuff and probably writes a really good program, but this book is accessible only to the experts, and even they won't be satisfied. The selection of topics covered is patchy; Patrick covers plenty of obvious headline subjects while ignoring lots of important but less-hyped ideas. The library project that runs through the book is dull, and captures the overall feeling of the book. The layout and diagrams make the presentation look clear and concise, but when you read what is actually on the page the analogies aren't helpful.

This well-intentioned attempt fails to help the beginner and the intermediate-to-expert VB programmer will find it tedious.

SUMMARY

- ☒ Single project used throughout
- ☒ Too much detail for beginners

SUPPLIER www.amazon.co.uk
ISBN 0321398009
PAGES 859

```
{
    sock.EndReceive(ar);
}
```

The use of a try block is to protect us from an error that becomes apparent only much later. It is possible that the main thread closes the socket before the OnReceive is complete, with the result that sock.EndReceive generates the error "Trying to access a disposed-of object". There is no way to test for an object being disposed of; the only solution is to use a try-catch block and handle the exception. This is another example of how things can become more complicated as soon as you have multiple threads.

We can start to unpack the data from the byte array and store it in the Dictionary. We need a two-element array into which we store the To address (four bytes starting at byte 16 in the buffer) and the total length of the packet (two bytes starting at byte 2):

```
UInt32[] temp = new UInt32[2];
temp[0] = BitConverter.ToUInt32(buf,
    16);
temp[1] = BitConverter.ToUInt16(buf,
    2);
```

The From address, 4 bytes starting at byte 12, is similarly stored in a suitable variable:

```
UInt32 From = BitConverter.
    ToUInt32(buf, 12);
```

Now we can store the array in the Dictionary using the From address as the key:

```
RawPackets[From] = temp;
```

If there is already an array stored under the same key (the same From address) it is replaced. To store several packets from the same IP address, you'll need to use a key that combines the From address with something else, such as a counter.

With all the data we want from the header processed, it's time to read another packet:

```
sock.BeginReceive(buf, 0,
    buf.Length,
    SocketFlags.None,
    new AsyncCallback(OnReceive),
    null);
```

This also ends the try block. The catch block is simply:

```
catch (ObjectDisposedException)
{ };
```

If the Socket object has been disposed of, we don't need to do anything else except avoid trying to access it, hence the existence of the empty brackets after the catch. The try-catch has an added advantage. If the main thread disposes of the Socket before it starts processing the data in the Dictionary, the try-catch will stop the Receive thread trying to update the Dictionary after the main thread starts this processing.

Finally, we need to put in the button's click event handler:

```
RawPackets.Clear();
```

This ensures that data from any previous scans isn't still in the dictionary.

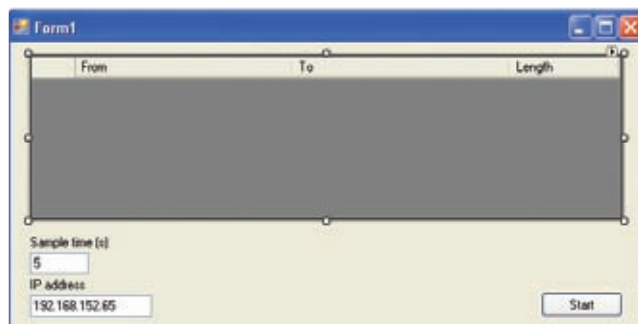
WORKING WITH TICK

One way or another, the scanning for packets comes to an end, either because the timer expires or the user clicks the Start/Cancel button. When it does, the Timer's Tick event handler is called. Its job is to shut down the socket and its associated thread, then get on with the job of unpacking the data. As there is no data collection going on, it can take its time as long as it doesn't make the user interface unresponsive. The Timer Tick event handler uses the same thread as the application, so while it is processing data nothing else happens.

There are many possible alternative ways of presenting the data. You could use a tree diagram to show the packet structure. In this project, we'll focus on who is sending and receiving packets, so a DataGridView control seems a good choice. Add this component to the form then add three columns: From, To and Length. The form should look like the example in the screenshot below.

We could simply unpack the data into the grid, but it is more useful to transfer it to a suitable 'struct' first:

```
struct Packet
{
    public IPAddress From;
    public IPAddress To;
    public Int32 Length;
```



◀ The packet sniffer's form layout

From	To	Length
193.194.158.108	195.168.252.65	25605
92.156.167.163	195.168.252.65	11264
70.42.134.12	195.168.252.65	10240
64.154.80.132	195.168.252.65	21248
207.46.109.65	195.168.252.65	12288

◀ The grid view shows who your computer has been talking to

```
};
```

The `IPAddress` class provides lots of useful methods for manipulating IP addresses. It has a constructor that can convert an IP address provided as an integer into the usual dotted string notation. You can add additional fields to the struct if you need more information from the IP header.

The `Timer Tick` event handler starts by closing the `Socket`. This has the effect of terminating the thread running the data collection and ensures that the main thread has exclusive access to the data. The button's caption is set back to `Start`, but it is also disabled because we don't want the user to start another scan while the data is being processed:

```
private void timer1_Tick(object sender, EventArgs e)
{
    sock.Close();
    timer1.Stop();
    button1.Enabled = false;
    button1.Text="Start";
}
```

At this point, we can clear the grid of any previous data:

```
dataGridView1.Rows.Clear();
```

To process each of the entries in the Dictionary, we'll use a `for-each` loop:

```
foreach (KeyValuePair<UInt32, UInt32[]> Add in RawPackets)
{
}
```

Each element in the Dictionary is represented as a `KeyValuePair` object and this is what is used as the index variable `Add` in the loop. Its properties `Key` and `Value` correspond to the key and the actual data. The data that `Add` contains is transferred to the new `Packet` struct:

```
Packet temp = new Packet();
temp.From = new IPAddress(Add.Key);
temp.To = new IPAddress(Add.Value[0]);
temp.Length = (Int32) Add.Value[1];
```

Apart from pulling the appropriate integers from the `Value` array, we've used the `IPAddress` constructor to perform a complex data conversion from an integer to an IP object.

The next task we have to perform is to transfer the data into the grid for display. There are so many ways to do this, it's difficult to know which is the best. One way, which emphasises the flexibility of the using objects and avoids

having to refer to row and column positions, is to first create individual cell objects initialised with the data:

```
DataGridViewTextBoxColumn CellFrom=
    new DataGridViewTextBoxColumn();
CellFrom.Value=temp.From;
DataGridViewTextBoxColumn CellTo =
    new DataGridViewTextBoxColumn();
CellTo.Value = temp.To;
DataGridViewTextBoxColumn CellLength =
    new DataGridViewTextBoxColumn();
CellLength.Value = temp.Length;
```

This creates three cells that aren't yet attached to a `DataGridView`. Next we create a row object and add the cells to it:

```
DataGridViewRow row = new
    DataGridViewRow();
row.Cells.Add(CellFrom);
row.Cells.Add(CellTo);
row.Cells.Add(CellLength);
```

Now we have a row that we can add to the grid:

```
dataGridView1.Rows.Add(row);
```

All that remains is to re-enable the button and let the user view the data:

```
}
button1.Enabled = true;
}
```

USING DNS

Most users don't recognise IP addresses, so we can improve `Sniffy` by looking up a host name (URL) for each IP address using the domain name system (DNS). If the computer isn't connected to a DNS server, this won't work. Adding this to the existing program is quick, but involves another very sophisticated use of threading. We can look up each `From` address by adding a single line to the end of the `For-each` loop:

```
Dns.BeginGetHostEntry(temp.From,
    GetHostEntryCallback, CellFrom);
```

This starts another asynchronous call, this time looking up the IP address contained in `temp.From`. All the work is going to be done by `GetHostEntryCallback`, a function that is called when the lookup completes. We'll write this below. We also pass the `CellFrom` object as the last parameter. You can pass any object in this way to any asynchronous call, and it will be passed to the callback function when the operation is complete. The first task our callback function has is extracting the cell object from the `IAAsyncResult` parameter passed to it:

```
public static void
    GetHostEntryCallback(
        IAsyncResult ar)
{
    DataGridViewTextBoxColumn Cell =
        (DataGridViewTextBoxColumn)ar.
        AsyncState;
```

This provides our function with the correct cell for the host name to be stored in without us having to keep track of which thread is working with which cell.

The host name can be retrieved using the `EndGetHostEntry(ar)` function, allowing us to update the cell's contents:

```
try
{
    Cell.Value = Cell.Value + " {" +
        Dns.EndGetHostEntry(ar).
        HostName + "}";
}
catch (SocketException){};
}
```

We need the `try-catch` because there are a range of error conditions that can arise while doing a DNS lookup. That's all you have to do to look up all the URLs in the grid. This goes well beyond the previous use of an asynchronous call, because now we are creating a separate thread for every `From` address in the grid. This technique is safe, because each thread gets its own cell object to work with and there is no possibility of two threads trying to work with the same one.

You might not believe it's this simple. What happens, for example, if the user clicks the `Start` button again before all the DNS threads have completed? It doesn't result in an application crash – but only because all the objects the DNS threads use are disposed of and this generates an exception, which is correctly handled. If you can avoid using multiple threads, it's simpler to do so. However in this case, and in the case of most communications programs, it's unavoidable.

`Sniffy` can be customised further to analyse network traffic so you can see exactly what your computer is doing, but that's a bigger project. ☐

From	To	Length
151.163.57.170 (151.163.57.170)	192.168.152.65	10240
92.156.167.163 (c529ca7a3.cable.wanadoo.nl)	192.168.152.65	10240
207.46.109.65	192.168.152.65	12288
62.194.65.84 (h65084.upc-h.chello.nl)	192.168.152.65	14848
212.72.38.145 (uk.statstat.com)	192.168.152.65	19712

◀ Looking up the `From` URLs makes it easier to see who is sending you data